Web Service Matchmaking based on Functional Similarity in Service Cloud

Kourosh Moradyan*, Omid Bushehrian a,*

*Department of Computer Eng. and IT, Shiraz University of Technology, Shiraz, Iran.

A R T I C L E I N F O.

Article history:
Received: 29 May 2015
Revised: 13 November 2015
Accepted: 02 March 2016
Published Online: 26 August 2016

Keywords:

A B S T R A C T

Nowadays increasing use of web as a means to accomplish daily tasks by calling web services, makes web services more and more significant. Users make a query on the Internet to find the required web service based on their needs. Cloud computing, due to its design and abundance of resources has become an ideal choice for web service providers to publish their services backed by cloud servers. The cloud can eliminate problems like web service availability and security. On the other side, obtaining the most relevant web service depends on user’s request accuracy and the mechanism used to match the request. Due to recent shutting down of public UDDI registries, most of web service matchmaking mechanisms are based on web service description files (WSDL) which are published on the owners’ websites. Semantic web services use OWL-S and WSMO instead of WSDL to describe services in a way that software agents are able to find appropriate services automatically. However, the high cost and effort needed to formally define web services makes this method impractical. In this paper we have proposed an ontology which formally models the user’s query for web services in the service cloud by considering both functional and syntactical dimensions. The stepwise matchmaking method of web services based on the user’s query is also presented. To show the precision of the proposed method, a set of experiments on a cluster of 3738 real web service WSDL documents has been performed.

1 Introduction

Service-Oriented Computing (SOC) is an emerging paradigm in which the ultimate goal is to realize development of distributed applications in heterogeneous environments. Services that can be defined as a piece of functionality done by an external provider are the basic building blocks. They are identified by an addressable interface describing their capabilities to the outer world. An application developed according to SOC paradigm includes a combination of service discovery, selection and invocation [1]. Cloud computing, due to its design, makes the use of cloud based web services trustable and hence service providers are eager to publish their services using cloud servers. Cloud guarantees availability, reliability, and quality of service [2]. A web service is described by means of the web Services Description Language (WSDL), and this description is published in a public Universal Description Discovery and Integration (UDDI) registry. Web
service communication is carried out through a form of XML messaging, such as Simple Object Access Protocol (SOAP) request or response. But major service providers such as Google, Amazon and Yahoo no longer publish their services in public UDDI registries; instead they publish them through their own websites. This trend forces users to discover web services using a search engine model. In [3] Al-Masri et al. showed that 53% of the registered services of UDDI Business Registry (UBR) are invalid, whereas 92% of services found by search engines are valid and active. Search engines process a user’s query to get the results back. So it is essential for the user to be aware of the correct keywords in order to retrieve the most relevant services that match the service request [4].

Cloud services provide a variety of services such as Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) paradigms [5]. The advances in SaaS paradigm has caused a rapid growth in the number of clouds and their services in recent years [6]. Some of the characteristics of SaaS such as high availability confidence of 99.99%, better scalability, cost saving, less maintenance and API integration have attracted service providers to migrate to cloud [5]. Daily increase of the number of web services has raised web service discovery as a new challenge in cloud [6].

Discovery is the process of matchmaking between user’s query and advertised service descriptions [7]. In this paper we have proposed an ontology (Figure 1) which formally models the user’s query for web services in the service cloud considering both functional and syntactical dimensions. Preparing user’s request using query ontology allows the users to verify the requests logically before giving it to a time consuming matchmaking algorithm. Due to informative form of requests based on query ontology there may be situations in which no service can satisfy the request. Based on this query language, a matchmaking method is also proposed and its accuracy is examined.

The rest of this paper is organized as follows. In Section 2 we briefly investigate the related works. Section 3 gives information about structure of WSDL documents. Section 4 explains our proposed query ontology. In this section we design an ontology based on information needed for requesting a service. In Section 5 we introduce our proposed approach. In Section 6 we explain the way to extract features from WSDL documents. In Section 7 the relation between user’s query and web services is established. Section 8 discusses the experiments results. And in Section 9 conclusions and future works will be discussed.

2 Related Works

Due to the importance of web services and the benefits that can be achieved from them, in recent years the web service technology has gained much attention. One of the most challenging problems is the discovery of web services based on their capabilities. Semantic web services are described using web ontology languages (OWL-S) [8] or Web Service Modeling Ontology (WSMO) [9]. Describing web services using such languages enables software agents to automatically discover, select, and use services as components in a composite that shapes an application. Semantic web services use ontologies to find matches between user requirements and service capabilities [10]. However, due to the high cost and effort needed to semantically describe web services, this approach has not been very widely applied. In fact non-semantic web services are more popular because they are supported by industry and also development tools [11]. Instead, non-semantic web services discovery approaches use information retrieval techniques [12].

In [13] matching of documents is carried out based on weighted keyword similarity that is a textual matching. They weigh each element in the WSDL document according to significance of that element in describing the desired service. Wu et al. in [14] proposed a system for improving web service discovery based on clustering and tagging text. In [15][16][17] matching is performed based on the semantics of schemas. They tried to find match between two schemas which is a linguistic analysis.

Many studies have been carried out to overcome the drawbacks of discovery techniques based on UDDI. In order to avoid the bottleneck, studies on distributed service discovery in peer-to-peer or grid environments have been carried out in [18]. Qiang et al. in [19] proposed a peer-to-peer-based decentralized service discovery approach that uses data distribution and lookup capabilities of Chord in order to discover services. They improved data availability by distributing descriptions of functionally alike web services to different successor nodes. Their approach, named Chord4S, also supports QoS-aware service discovery. Dasgupta et al. in [20] proposed a multi-agent based distributed platform for efficient semantic service discovery. They also have addressed the scalability problem of discovery approaches.

Abundance of Web services with the same functionality over the Internet and the daily additions has become a problem that causes longer response time for matchmaking algorithms. In order to overcome this problem some of the approaches consider non-functional aspects of web services to filter and select more accurate services. Kritikos et al. in [21] proposed
three techniques that manages space of offered services to improve the overall matchmaking time. Suraci et al. in [22] proposed a distributed workload control algorithm to manage the requests. They concentrated on minimizing the overall latencies that requester agents would face.

Service-based applications may need to replace the services that are no longer available or fail to meet certain requirements during the execution of application. Zisman et al. in [23] proposed a framework that supports runtime service discovery.

Cassar et al. in [24] presented a method, that using probabilistic machine learning techniques, is capable of extracting hidden features from semantic web service description documents such as OWL-S and WSMO. They used these features to build a model that represents different types of service descriptions in a vector which enables heterogeneous service descriptions to be represented, compared or discovered.

The majority of web services exist without explicit associated semantic descriptions that leads to not being considered as relevant to specific user’s request during web service discovery. Paliwal et al. in [25] tried to categorize web service descriptions semantically and also enhance semantics of user’s request.

Semantic web services, as the most revolutionary technology, promises to enable machine to machine interaction on the web and to automate the discovery and selection of the most suitable web services in building service based applications. But most of the web service descriptions are using conventional web service annotations, e.g. WSDL that lacks semantic annotations. Farrag et al. in [26] proposed a mapping algorithm that facilitates redefinition of conventional web service annotations using semantic annotations such as OWL-S.

Sangers et al. in [27] proposed a semantic web service discovery framework in order to find services using natural language processing techniques. They used part-of-speech tagging, lemmatization, and word sense disambiguation techniques to match keywords from the user’s request with the web service descriptions given in WSMO. Klusch et al. in [28] proposed a hybrid semantic web service matchmaker, called OWLS-MX, that improves logic-based semantic matching of OWL-S services by leveraging non-logic-based information retrieval techniques. In [29] the analysis of experimental evaluations of performance of OWLS-MX is summarized and an improved matchmaker is implemented.

In some QoS-based approaches in web service dis-
covery and selection, it is assumed that the QoS data expressed by service providers are trustworthy and effective. However, the values of QoS attributes may be unreliable. El-Kafrawy et al. in [30] proposed a web service discovery and selection model that considers trust factor when calculating web service reputation. Lin et al. in [31] proposed a mechanism in which web service discovery is achieved in two phases based on QoS and collaborative filtering. Their mechanism verifies QoS values stated by service providers. Sunchithra et al. in [32] used a third party between provider and consumer to check for integrity of QoS values and calculate some implicit metrics. In [33] web service discovery is carried out by considering four similarity assessments to match services. They used WordNet and HowNet to calculate the distance between two words in a hierarchy concept.

3 WSDL Document Structure
WSDL is a language based on XML to describe non-semantic web services. It provides descriptions about the operations that a web service can perform as well as its location. Inside the root element of a WSDL file, services are defined using six major components [34]:

- `<types>` describes a container for defining data types using some type system such as XML Schema Definition (XSD).
- `<message>` represents an abstract data that is transmitted. A message consists of logical parts, each of which is associated with a definition within some type system.
- `<portType>` is a set of abstract operations. Each operation refers to an input message and/or output message.
- `<binding>` specifies concrete protocol and data format specifications for the operations and messages defined by a particular portType.
- `<port>` specifies an address for a binding, thus defines a single communication endpoint.
- `<service>` aggregates a set of related ports.

4 Query Ontology
We proposed an ontology (Figure 1) to formally define the user query for service matchmaking. The ontology structure states that every query is a workflow (composition) of one or more tasks that the user requires. Each task in the query has a name and a description that describes its function using natural language comments. Those comments include some essential keywords, as well as parameters that are categorized into input parameters and output parameters. For every task parameter, its type should be mentioned. Parameter name is also considered because it may contain useful words which are relevant. We also consider Quality of Service (QoS) criteria as an option that the user can consider. We created a tool for creating queries based on the ontology. User’s query is stored as RDF statements for further processing.

As an example, consider the user requests for a task to check weather information for a city, so he/she prepares his request using query creating tool. Request is a web service that is capable of retrieving weather information for a city that the user enters as a string. Request is stored as RDF statements to be processed next. In order to transform the user query to RDF statements we use query ontology axioms to prepare instance statements by creating individuals from ontology resources and then establishing relationships between individuals using predefined predicates in ontology. In definition of concepts in ontology we set constraints which allow us to check them on newly added statements. We use textual data in a request as literal values of statements. After transformation, in request process phase reasoning over RDF statements is done and the consistency among statements is checked. If no inconsistency is found, it means user’s request is logically correct and is ready to be delivered to a web service discovery system. We use Pellet reasoner in Jena to check the consistency of user request. We first load the query ontology in a model and then add RDF statements created from user request to it. Reasoning is done over the model which contains query ontology schema and also newly added statements from the user request. In Figure 2 a fragment of RDF file containing user information is presented. The steps for preparing the request are illustrated in Figure 3.

5 Matchmaking Approach
Matchmaking process is carried out in three phases. First functional matching is done that matches services functionally, then the syntactical matching of services with the same functional capabilities is done and at the end QoS matching of services is considered, i.e. to find services with the required QoS capabilities according to the user expectations. Figure 4 shows the sequence of matchmaking process. We use the information available in WSDL document to perform matchmaking. We parse the WSDL document to retrieve features that describe the function of the web service semantically. Features of interest are service name, operation names, operation documentations, parameter names and types, and service documentation. We use root element, service element, types element, message element, and portType element to extract our required features. These are the features used to compare user’s query with web services.
In fact the logic of our matchmaking algorithm is based on this reasoning that when user describes his required task, he uses some keywords to explain it. Obviously the keywords are nouns and verbs of the sentences (just content words, not function words) and we try to find synonym matches for these keywords among the keywords extracted from documents used to describe web services. In case the services are not annotated with documentations, only service, parameter and operation names are used to extract the...
keywords.

By comparing the user’s query content with the extracted features semantically, services with the required functionality are matched to the user’s query. Next we compare matched services to the user’s query according to the operation signatures. To match the signatures, we consider the number and the types of the input and output parameters for the request and the service. Finally we match the QoS criteria issued in the request to the QoS parameters in the service. So the most relevant web services are returned. Figure 5 shows the proposed matchmaking method.

6 Extracting Features

In order to match the query and the services, we need to extract information from the user query and WSDL documents. The required information are: task names, task descriptions, Input and Output parameter names and types from the user query, and the QoS values specified in the request as well as service name, service documentation, operation names, operation documentation, parameter names and parameter types of the input and output parameters for every operation in WSDL documents. We process this information to retrieve the required features.

A. Processing user query

A user query is a collection of RDF statements that is queried using SPARQL. To do so we use the Jena API[1] and we make some query to extract information from the user request. Features that we need are:

(1) **User’s description for the task.** We process this description using OpenNLP API. Processing description is as follows:
   a) **Extracting sentences.** We split text into sentences.
   b) **Tokenizing every sentence.**
   c) **Tagging every token.**
   d) **Extracting keywords using tags.** We extract just words that are tagged as noun or verb, so function words are not retrieved.

---

[1] https://jena.apache.org/
e) **Preprocessing keywords.** We preprocess keywords to remove the synonym keywords from the list of obtained keywords.

(2) **Task names.** Every task name is comprised of some meaningful words that states the function of the task.

(3) **Signature of the required tasks.** This is the information about the name, the number and the types of the input and the output parameters per task.

(4) **Quality of Service parameters.** We use these parameters from the user request to compare them with the corresponding values which were calculated previously for every service.

**B. Parsing WSDL documents**

We need some essential features in WSDL documents that explain the function of the corresponding web service. Features like service name, service documentation, operation names, operation documentations and parameter names. We also extract operation signatures in order to compare to the user’s annotations and parameter names. We also extract operation documentations in order to compare to the user’s annotations.

In order to parse WSDL document we use Membrane SOA Model\(^2\) that is a Java API for WSDL and XML Schema. Parsing is as follows:

(1) **Extracting and processing Web service documentation.** Web service documentations in a WSDL document may be repeated and also may contain some useless information like URL and contact number. In order to remove this information we use Regular Expressions in Java by accommodating their patterns to recognize them inside the text and then removing them. Then we process the description as

a) **Extracting sentences.**

b) **Tokenizing every sentence.**

c) **Tagging every token.**

d) **Extracting keywords using tags.** We extract just the words that are tagged as noun or verb, so function words are not retrieved.

e) **Preprocessing keywords.** Like preprocessing the keywords obtained from the request we also preprocess these keywords to remove the synonym keywords from the list of obtained keywords.

(2) **Extracting and processing service name, operation names and parameter names.** Service name, operation names and parameter names in WSDL documents follow the Camel Case naming convention, so we use the Camel Case writing pattern to split names into their formative words. These extracted words may contain non-letter characters at the beginning or at the end of the word, so we check them to delete extra characters from them. Processed words are checked to be meaningful in English and so the words that are special names are ignored and the others are added to a list to be compared to user’s keywords semantically. In order to check that a word is meaningful or not we use WordNet. We use JAWS\(^3\) API to determine the synonym set of a given word. If the returned synonym set is empty, it means that the given word is not an English word. Keywords gathered from operation names and parameter names are added to a separate vector to be compared to keywords gathered from task names and their parameters from user request.

Service names try to describe the overall functions of services, so we add the keywords that are extracted from the service name to the vector of keywords extracted from web service document.

(3) **Extracting and processing operation documentation.** Every operation may have a documentation that describes its function. We extract this documentation and do the same work as presented in step 1. We collect the keywords from this documentation and web service documentation in a list to be processed next.

(4) **Extracting signatures of operations.** We extract the types of the input parameters as well as the output parameters to compare with user query’s signature(s).

**7 Established Realizations**

Assume that the user’s query consists of \(n\) tasks \(T = \{t_1, t_2, \cdots, t_n\}\). The similarity measure \(Similarity(t_i, s_j)\) between task \(t_i\) from the user request and service \(s_j\) is as follows:

\[
Similarity(t_i, s_j) = \Phi_1 S(W_t, W_s) + \Phi_2 Simname(t_i, s_j) + \Phi_3 Matchsignature(t_i, s_j) + \Phi_4 MatchQoS(t_i, s_j) \quad (1)
\]

In Equation (1), \(\Phi_1 = \Phi_2 = \Phi_3 = \Phi_4 = 0.25\) are weights of the assessment methods and \(W_t\) and \(W_s\) are keywords gathered from the user query documentation of task \(t_i\) and the keywords of the embedded documentations and the service name in service \(s_j\). respectively. If \(W_t\) is the intersection of \(W_t\) and \(W_s\) computes the similarity between two list of gathered keywords and is calculated as follows:

\[
S(W_t, W_s) = \frac{|W_t \cap W_s|}{|W_t|} \quad (2)
\]
In Equation (2), \( |W_t \cap W_j| \) is the cardinality of the set of common keywords in \( W_t \) and \( W_j \) that semantically are the same and is calculated as follows:

If \( W_t = \{w_1, w_2, \ldots, w_n\} \) contains \( n \) keywords and \( W_j = \{w_1, w_2, \ldots, w_m\} \) contains \( m \) keywords, then if we define \( CK \) as the set of common keywords and \( \text{Synset}(x) \) as the set of synonyms of keyword \( x \), then the relationship defined in (3) is established:

\[
CK \equiv \{ w \in (W_t \cap W_j) | w \in W_t \land \exists x \in W_j : w \in \text{Synset}(x) \}
\]

In Equation (4), \( \text{Sim}_{name}(t_i, s_j) \) computes the similarity between the names of task \( t_i \) and the operation names of service \( s_j \) and their parameter names. We calculate this similarity using Equation (4) as follows:

\[
\text{Sim}_{name}(t_i, s_j) = \max_{k=1 \ldots n} \left( S(TN_i, O_{jk}) \right) \tag{4}
\]

In which, \( TN_i \) is a list containing the name of task \( t_i \) and its parameters split into meaningful words, and \( O_{jk} \) is a list of the name of \( k \)th operation in service \( s_j \) and its parameters split into meaningful words.

In order to match the signature of user query task \( t_i \) with an operation signature in \( s_j \), we use \( \text{Match}_{signature}(t_i, s_j) \) in Equation (5). We check signatures according to the number and types of input parameters and output parameters. In Equation (5), we show how to match signatures as follows:

\[
\text{Match}_{signature}(t_i, s_j) = \max_{k=1 \ldots n} \left( \text{matching}(t_i, o_{jk}) \right) \tag{5}
\]

In which, \( o_{jk} \) is \( k \)th operation in service \( s_j \) and \( t_i \) is the user’s required task. And \( \text{matching}(t_i, o_{jk}) \) is calculated as follows:

\[
\text{matching}(t_i, o_{jk}) = \frac{M(t_i, o_{jk})}{\text{avg}(E(t_i), E(o_{jk}))} \tag{6}
\]

Where, \( M(t_i, o_{jk}) \) is the number of matched types between \( t_i \) and \( o_{jk} \) and \( E(t_i), E(o_{jk}) \) are the total number of types in \( t_i \) and \( o_{jk} \) respectively.

Quality of Service (QoS) refers to non-functional attributes of web services. Main attributes are as follows [33]:

- **Availability**: it is the quality aspect of service which concerns about presence and being ready for immediate use of web service. Larger values indicates higher availability.
- **Response Time**: it is measured as the time a service takes to respond to requests.
- **Throughput**: it is defined as the rate at which a service is able to process requests.
- **Reliability**: it is a quality aspect of a web service defined as the probability that the web service can respond to its users without failure.

- **Security**: comprises the existence and authentication mechanisms offered by a service.

In Equation (7), we calculate \( \text{Match}_{QoS}(t_i, s_j) \) using the following formula:

\[
\text{Match}_{QoS}(t_i, s_j) = \sum_{i=1}^{n} \omega_i \text{sim}_{A_i}(t_i, s_j) \tag{7}
\]

In Equation (8), \( n \) equals to the number of considered QoS attributes, \( A_i \) is the \( i \)th attribute and \( \omega_i = \frac{1}{n} \) is the weight of given attribute. Method \( \text{sim}_{A_i}(t_i, s_j) \) calculates the amount of similarity between attribute \( A_i \) in task \( t_i \) and service \( s_j \) and for all of attributes is defined as follows [33]:

\[
\text{sim}_{A_i}(t_i, s_j) = \frac{1 - |t_i.A_i - s_j.A_i|}{2 - t_i.A_i - s_j.A_i} \tag{8}
\]

For example if we only consider response time and throughput attributes then:

\[
\text{Match}_{QoS}(t_i, s_j) = \sum_{i=1}^{2} \omega_i \text{sim}_{A_i}(t_i, s_j)
\]

\[
= 0.5 \times \text{sim}_{\text{responseTime}}(t_i, s_j) + 0.5 \times \text{sim}_{\text{throughput}}(t_i, s_j) \tag{9}
\]

In which:

\[
\text{sim}_{\text{responseTime}}(t_i, s_j) = \frac{1 - |t_i.\text{responseTime} - s_j.\text{responseTime}|}{2 - t_i.\text{responseTime} - s_j.\text{responseTime}} \tag{10}
\]

And

\[
\text{sim}_{\text{throughput}}(t_i, s_j) = \frac{1 - |t_i.\text{throughput} - s_j.\text{throughput}|}{2 - t_i.\text{throughput} - s_j.\text{throughput}} \tag{11}
\]

A. Case study

Consider user requests for a task in a service to calculate summation of two numbers using a method called sum that gets two float arguments as input and a float as result, with a description as “adds two numbers.” and some QoS expected. In order to calculate \( \text{Similarity}(t_i, s_j) \) first the content words are extracted from the description which includes “adds” and “numbers” are extracted. These words are used to be matched against the content words previously extracted from web service documentations. The amount of semantic matching is calculated as \( S(W_t, W_j) \). Then the word “sum” as a task name is processed and is compared semantically to every operation name in the given web service. This value calculates \( \text{Sim}_{name}(t_i, s_j) \). At the end, the amount of matching between task signature and operation’s signatures \( \{\text{Match}_{signature}(t_i, s_j)\} \) and also \( \text{Match}_{QoS}(t_i, s_j) \) are calculated by comparing the types of the input and the output parameters.
and matching QoS attributes. By assigning the same weight to every similarity measure, the total similarity, \( \text{Similarity}(t_i, s_j) \), for every web service to user request is calculated.

8 Experiments and Results

Our experiments are based on a benchmark of 3738 real (online) web services collected from all over the world [36]. In this benchmark the QoS of response time and throughput attributes have been calculated. We manually classified web services into calculator, email validation, SMS, and weather categories. We have done two series of experiments.

Our experiments are based on sixty five manually classified WSDL documents from the mentioned benchmark shown in Table 1. We identified service descriptions from four categories: calculator (twenty two services), email validation (six services), SMS (twenty five services), and weather (twelve services). In this section a set of experiments are reported: our matchmaker implemented with four similarity assessment methods. Summation of the values gathered from Equation (2) and Equation (4) calculates the amount of functional similarity between the request and services. The third method in Equation (5) calculates the amount of signature matching between the request and services. And Equation (7) filters the proposed services to match the user request more accurately.

To evaluate the effectiveness of our proposed matchmaker we used precision and recall metrics. “Precision is the proportion of retrieved documents that are relevant, and recall is the proportion of relevant documents that are retrieved” [33]. We used the following Equations:

\[
\text{Recall} = \frac{|\text{true positive}|}{|\text{true positive}| + |\text{false negative}|} \quad (12)
\]

\[
\text{Precision} = \frac{|\text{true positive}|}{|\text{true positive}| + |\text{false positive}|} \quad (13)
\]

Figure 6 shows the results of our experiments. We ran different queries over four categories to obtain these results. The web services were given similarity values to the request. In our experiments we considered services whose similarities to the request were bigger than 30%. This threshold was obtained empirically as it fitted our experiments well.

We compared our approach to a similar one. We implemented the approach in [37] and performed our experiments on sixty five WSDL documents, mentioned above. Experiments showed our approach has better performance.

In [37], the proposed approach focuses on finding web services that have exactly the same name and description and web services with the same function but described using different synonym words have less importance. Also they have used hierarchical structures between words that relate them with irrelevant functional semantics. For example words like “get” and “add” shouldn’t be compared.

In our experiments we can see that in some cases, similarity values are rather low. In our opinion, the reason may be some faulty information in WSDL documents, for example an operation named “calcAdd” is split to “calc” and “Add” and because “calc” is meaningless in English, it is deleted. Or for another example in a WSDL document that have operations named “add”, “sub”, “mult”, and “divide”, the operation names “sub” and “mult” are discarded because they are meaningless. By finding the best synonym words and replacing them with the abbreviated words, as a solution to these problems, the results will be improved.

We also compared our approach to goDiscovery [11], a non-semantic web service discovery approach based on WSDL which uses statistical methods and indexing techniques to improve precision and response time of discovery process. The same benchmark as ours has been used to evaluate goDiscovery.

In order to compare our work we use F-score measure that is a measure to evaluate the overall system accuracy. F-score is a weighted average of Precision (P) and Recall (R) and is calculated as follows [11]:

\[
F\text{-score} = 2 \cdot \frac{P \cdot R}{P + R} \quad (14)
\]

Figure 7 shows the F-score for our approach, goDiscovery [11] and approach in [37]. Results show that our approach outperforms the other approaches according to the overall accuracy. In our approach we focus on matching services semantically with the same functionality. We consider both exact and functionally alike results to the request. In [37], the focus is on finding the exact matches with the request. Therefore some of the potential results are not considered. goDiscovery is not a semantic-based approach but a statistical and indexing approach that is based on the retrieved information of current document. It uses them to build its model.
<table>
<thead>
<tr>
<th>Category</th>
<th>WSDL URI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Email Validation</td>
<td><a href="http://webservices.data-8.co.uk/EmailValidation.asmx?WSDL">http://webservices.data-8.co.uk/EmailValidation.asmx?WSDL</a></td>
</tr>
<tr>
<td>SMS</td>
<td><a href="http://www.smsservice.com/portal/080/SMSateway.asmx?wsd1">http://www.smsservice.com/portal/080/SMSateway.asmx?wsd1</a></td>
</tr>
<tr>
<td>Weather</td>
<td><a href="http://www.thinkpage.cn/weather/WeatherService.asmx?wsdl">http://www.thinkpage.cn/weather/WeatherService.asmx?wsdl</a></td>
</tr>
<tr>
<td></td>
<td><strong>Table 1. Manually Classified Web Services</strong></td>
</tr>
<tr>
<td>Category</td>
<td>WSDL URI</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>Calculator</td>
<td><a href="http://www.vejlebib.dk/webservices/mathservice.asmx?WSDL">http://www.vejlebib.dk/webservices/mathservice.asmx?WSDL</a></td>
</tr>
<tr>
<td>Email Validation</td>
<td><a href="http://www.smsservice.com/portal/080/SMSateway.asmx?wsd1">http://www.smsservice.com/portal/080/SMSateway.asmx?wsd1</a></td>
</tr>
<tr>
<td>SMS</td>
<td><a href="http://www.smsservice.com/portal/080/SMSateway.asmx?wsd1">http://www.smsservice.com/portal/080/SMSateway.asmx?wsd1</a></td>
</tr>
<tr>
<td>Weather</td>
<td><a href="http://www.thinkpage.cn/weather/WeatherService.asmx?wsdl">http://www.thinkpage.cn/weather/WeatherService.asmx?wsdl</a></td>
</tr>
</tbody>
</table>
9 Conclusion and Future Work

In this paper we make complementary contributions to web service matchmaking. We design an ontology to formulate a user’s query structure. To do so, we propose a Query Ontology that persuades users to prepare a structured request with full information needed to do matchmaking. It also verifies the user request logically before delivering it to matchmaking process which is a time consuming process. So requests with no answer are discarded before processing, which helps matchmaking algorithms to process only appropriate requests and save time. We also propose a semantic-based matchmaking algorithm based on WSDL documents to discover services based on their functionalities. Service matchmaking categorizes web services into four categories and for each category a similarity assessment method is given. Experimental results show that our approach has an improved precision in retrieving services in comparison to previous approaches.

It is hard to push programmers to use a formal specification for each web service they write. There are problems like respecting Camel Case writing convention used by programmers, using abbreviated words in defining web services, and considering relationship between words which belong to a specific domain. In the future we plan to conquer such problems and obstacles in our matchmaking method to get better results.

References

[1] Debajyoti Mukhopadhyay and Archana Chougule. A survey on web service discov-


[34] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (wsdl) 1.1, 2002.


[37] Yiqiao Wang and Eleni Stroulia. Semantic

Kourosh Moradyan received his Associate Degree in computer software from University of Kurdistan, Sanandaj, IRAN and his B.Sc. degree in software engineering from Zagros Institute of Higher Education, Kermanshah, IRAN. He also received his M.Sc. degree in software engineering from Shiraz University of Technology, Shiraz, IRAN. His research interests include Semantic Web, and Web Service Discovery.

Omid Bushehrian received the BSc in software engineering from AmirKabir University of Technology (Tehran Polytechnics), IRAN, the MSc and Ph.D degrees in software engineering from the University of Science and Technology, IRAN. He is currently a faculty member at Shiraz university of Technology, IRAN. His main research interest is cloud computing and software quality.